



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

(Laboratorio di)  
**Amministrazione di sistemi**

# **Gestione dei pacchetti software**

**Marco Prandini**

Dipartimento di Informatica – Scienza e Ingegneria

# Gestione del software

## ■ Ciclo di vita

- installazione
- aggiornamento
- disinstallazione

## ■ Problematiche

- prerequisiti hardware/s.o.
- dipendenze da/di altri componenti software
- configurazione

# Installazione manuale

## ■ Da binari

- semplice copia nei "posti giusti"
- verifica manuale della compatibilità con l'architettura
- verifica manuale del soddisfacimento delle dipendenze

## ■ Da sorgente

- necessità di compilazione
- indipendenza dall'architettura
- possibile maggior flessibilità nel soddisfacimento delle dipendenze

# Installazione manuale

## ■ Dipendenze del componente software da altri

- Nel caso di un'installazione da binari, probabile necessità di disporre non solo dei software indicati come prerequisiti, ma anche che essi siano di una versione specifica
- Nel caso di installazione da sorgente, qualche grado di flessibilità (possibilità che i sorgenti dispongano di diverse interfacce per adeguarsi a cosa si trova sul sistema)
  - Necessità di disporre non solo dei componenti runtime relativi ai software richiesti, ma anche delle librerie di sviluppo (prototipi, interfacce, librerie per collegamento statico, ...)
    - In un sistema “ideale” ho tutti i sorgenti per cui dispongo sempre di tutti questi elementi
    - Nelle distribuzioni, per flessibilità, ogni pacchetto software ha un corrispondente pacchetto -dev o -devel (vedi prossime slide)

# Installazione manuale tipica in Linux

- Il caso più comune è quello di software
  - distribuito per mezzo di un archivio tar.gz
  - scritto in C
  - predisposto alla compilazione tramite autoconf
    - verifica se sono soddisfatti tutti i prerequisiti
    - rileva le versioni ed le collocazioni dei pacchetti sul sistema
    - accetta dall'utente la specifica di varianti (attivazione/disattivazione di funzionalità, preferenze architetturali, ...)
    - genera i Makefile sulla base delle specificità del sistema e delle scelte operate dall'utente



# Installazione manuale tipica in Linux

- I passi tipici quindi sono:
  - reperimento del software
  - estrazione del pacchetto
  - esame delle scelte disponibili
  - configurazione dei sorgenti
  - compilazione
  - installazione
- **NOTA:** solo quest'ultima operazione può richiedere i diritti di superutente, e quindi si deve evitare di compiere le precedenti come *root*. Sono noti casi di malware che sfruttano proprio la cattiva abitudine di eseguire una o più delle operazioni preliminari con diritti eccessivi.

# Autenticazione

- La prima cautela da usare quando si scarica software dovrebbe essere quella di verificarne l'autenticità da una firma digitale
- Naturalmente per verificare una firma serve una chiave pubblica fidata
- Es:
  - 1) `gpg --verify FILE.asc FILE.tar.gz`
    - (mostra il key id)
  - 2) `gpg --keyserver pgpkeys.mit.edu --recv-key <KEY_ID>`
    - l'autenticità della chiave in questo caso deriva solo dalla fonte... basta?
    - valutare caso per caso
    - seguire indicazioni specifiche <https://httpd.apache.org/dev/verification.html>
  - 3) ripetere il passo (1)
- Come minimo, dovrebbe essere disponibile un fingerprint
- Basta mettere il fingerprint (es. in formato .sha256) nella stessa directory del file da verificare e lanciare  
`sha256 -c FILE.sha256`
- esempio di fonte che li offre entrambi
  - <https://httpd.apache.org/download.cgi>

# Installazione manuale tipica in Linux

## ■ estrazione del pacchetto

- solitamente si presenta come archivio tar compresso
- è buona prassi determinare una collocazione sensata per i sorgenti ed estrarre in tale directory l'archivio
  - nel caso si stia per affrontare un upgrade sostanziale del sistema, che coinvolga numerose applicazioni, può essere utile raccogliere in modo più chiaro tutti i pacchetti che verranno installati unitariamente
- è prudente testare l'archivio prima dell'estrazione per verificare la gerarchia di directory che genera
- Es: 

```
cd /usr/local/src  
tar tvzf net-snmp-5.4.tar.gz  
tar xvzf net-snmp-5.4.tar.gz
```



# Installazione manuale tipica in Linux

- esame delle scelte disponibili
  - si entra nella directory generata dall'estrazione e si esamina il contenuto
    - è bene leggere i file README ed INSTALL che di solito accompagnano il software
  - se esiste un eseguibile di nome **configure** lo si lancia con il parametro **--help** per ottenere la lista dei parametri di configurazione disponibili
    - scelte comuni riguardano la collocazione del software, l'attivazione o la disattivazione di sottocomponenti, la predisposizione dei componenti attivati come moduli dinamicamente caricabili piuttosto che la loro integrazione statica nel codice, ...

# Installazione manuale tipica in Linux

## ■ configurazione dei sorgenti

- si lancia nuovamente **configure** con i parametri scelti
- si risolvono i problemi evidenziati da configure (tipicamente assenza di pacchetti necessari come prerequisiti)
  - configure non è a prova d'errore, può servire un'indagine manuale a volte complessa
  - indicazioni utili (spesso indispensabili) sono nei file README e INSTALL
  - la procedura genera un `config.log`

## ■ compilazione

- si lancia **make** o si seguono le indicazioni presenti nell'output generato dal passo precedente

## ■ installazione

- si lancia **sudo make install**

# Riflessioni sull'installazione da sorgenti

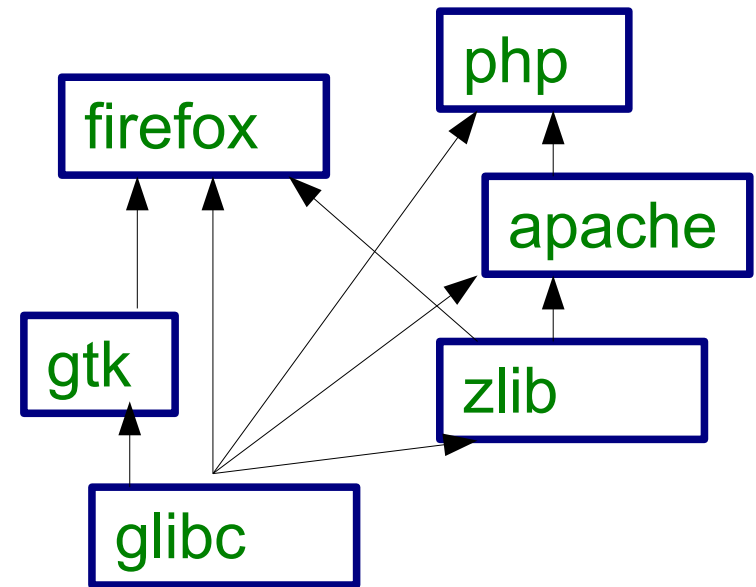
- **Offre la possibilità teorica di verificare il codice.**
  - Se non lo fate → falso senso di sicurezza.
  - Se vi fidate della firma sull'archivio, non è diverso che verificare la firma su di un binario.
- **Più difficile da mantenere**
- **Richiede MOLTI componenti ausiliari**
  - Header e librerie
  - Processori di macro
  - Compilatori e linker ...
  - sono tutti elementi che possono avvantaggiare un attaccante

<https://wiki.c2.com/?TheKenThompsonHack>
- **Entrano in scena le distribuzioni e i pacchetti:**
  - Chiavi di verifica installate una volta per tutte
    - comodo ma SPOF
  - Gestione automatica delle dipendenze
  - Garanzia di compatibilità binaria tra tutti gli elementi del set

# Installazione assistita

- Comunemente effettuata per mezzo di software ausiliari
  - package manager specifico della distribuzione Linux (rpm/yum, dpkg/apt, ...)
  - installer per Windows
- Un tool di installazione
  - può farsi carico delle verifiche relative alle dipendenze
  - non può configurare ogni dettaglio del sistema in modo specifico
  - può generare dinamicamente dati specifici

Esempio di *grafo delle dipendenze*:



A → B significa che A “serve” per B; “serve” può essere una dipendenza tra funzionalità logiche (non ha senso avere un linguaggio di generazione pagine web senza un web server) o fisiche (un binario linkato dinamicamente non gira senza tutte le librerie di cui importa i simboli)



# Pacchetti

- Le *distribuzioni* di Linux organizzano il software in *pacchetti* e dispongono di un *package manager* per la loro gestione
- Un pacchetto si presenta sotto forma di singolo file che contiene in forma compatta l'insieme di
  - software precompilato
  - criteri per la verifica della compatibilità e dei prerequisiti
  - procedure di pre/post-installazione
- La garanzia della compatibilità con un determinato sistema può essere data solo a patto di vincolare con precisione alcuni parametri:
  - architettura
  - versione della distribuzione
  - versione del software contenuto nel pacchetto

# Distribuzioni: criteri per la scelta

## Architetture supportate

- Tutte le distribuzioni supportano i processori Intel 32bit, la maggior parte quelli a 64bit, alcune sono disponibili per tutte le varietà di processori su cui è stato portato il kernel
- È bene ricordare che i pacchetti di terze parti potrebbero non essere disponibili per tutte le architetture supportate

## Stabilità vs. Aggiornamento

- Il processo di rilascio frequente e continuo del software nel mondo GNU/Linux ha come conseguenza inevitabile che le versioni più aggiornate possano essere meno stabili
- Vi sono distribuzioni che hanno come filosofia l'inclusione dei pacchetti più recenti (e quindi con funzionalità maggiori) anche a costo di una minor robustezza, ed altre che garantiscono l'inclusione solo di software ben collaudato

# Distribuzioni: criteri per la scelta

## Version vs rolling

- Alcune distribuzioni sono “versionate”: durante il ciclo di vita di una versione vengono forniti solo aggiornamenti correttivi, tutte le novità vengono testate e accumulate per la pubblicazione in una nuova versione (che va installata sovrascrivendo la precedente)
- Altre sono “rolling”: ogni volta che c'è una novità viene testata e distribuita, quindi in ogni momento il sistema è alla versione più recente

## Supporto e durata

- La disponibilità di supporto garantito è tipica delle distribuzioni commerciali, ma anche con le distribuzioni gratuite più diffuse, in virtù della dimensione della relativa comunità di utenti, è semplice risolvere eventuali problemi
- Per installazioni di tipo server esistono varianti denominate LTS (Long Term Support): per 5/7 anni chi cura la distro garantisce che gli aggiornamenti non modifichino le API (tipicamente viene garantito solo il backporting dei security fix, non quello di tutti i bug fix)

## Ampiezza del set di pacchetti

- Si va dai 1500 delle distro minimali ai 26000 di Debian
- Una scelta intelligente mette tutto l'essenziale in 1 CD

# Debian e Red Hat

- Due distribuzioni capostipite da cui sono state derivate quasi tutte le varianti più diffuse  
<http://upload.wikimedia.org/wikipedia/commons/9/9a/Gldt1009.svg>
- Due sistemi di gestione dei pacchetti con molte somiglianze
  - Tool di basso livello per la gestione dei singoli pacchetti
  - Tool intermedi per la gestione coordinata di pacchetti e dipendenze
  - Tool per il reperimento automatico da *repository* dei pacchetti necessari



# Pacchetti

- I pacchetti per le distribuzioni Debian e derivate (es. Ubuntu) sono in formato *.deb*

– **aptitude**-**0.2.15.9**-**2**\_**i386**.deb

nome

versione del software

architettura

versione del pacchetto

- I pacchetti per le distribuzioni RedHat e derivate (es. CentOS, Fedora) sono in formato *.rpm*

– **httpd**-**2.4.6**-**45**.el7.centos.**x86\_64**.rpm

# Una nota per gli sviluppatori: pacchetti base e development

## ■ zlib1g

- /usr/lib/libz.so.1.2.3.3

per ogni funzione, es. *compress*:

codice oggetto in formato adatto per il  
linking dinamico

## ■ zlib1g-dev

- /usr/lib/libz.a

codice oggetto in formato adatto per il  
linking statico

- /usr/include/zconf.h

- /usr/include/zlib.h

prototipo per il compilatore

- /usr/include/zlibdefs.h

Con questa suddivisione si risparmia (molto) spazio sui sistemi che non sono usati per *sviluppare* codice basato su questa libreria, nei quali il primo pacchetto fornisce da solo il necessario per *usare* codice già pronto in forma binaria che referencia le funzioni della libreria

■ Su sistemi *deb* → pacchetti “-dev”

■ Su sistemi *rpm* → pacchetti “-devel”

# Esempio di verifica delle dipendenze dinamiche

## ■ **ldd /usr/sbin/sshd**

linux-gate.so.1 => (0xffffe000)  
libwrap.so.0 => /lib/libwrap.so.0 (0xb7ef7000)  
libpam.so.0 => /lib/libpam.so.0 (0xb7eed000)  
libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7ee8000)  
libselinux.so.1 => /lib/libselinux.so.1 (0xb7ed2000)  
libresolv.so.2 => /lib/tls/i686/cmov/libresolv.so.2 (0xb7ebf000)  
libcrypto.so.0.9.8 => /usr/lib/i686/cmov/libcrypto.so.0.9.8 (0xb7d7c000)  
libutil.so.1 => /lib/tls/i686/cmov/libutil.so.1 (0xb7d78000)  
**libz.so.1 => /usr/lib/libz.so.1 (0xb7d63000)**  
libnsl.so.1 => /lib/tls/i686/cmov/libnsl.so.1 (0xb7d4a000)  
libcrypt.so.1 => /lib/tls/i686/cmov/libcrypt.so.1 (0xb7d1c000)  
libgssapi\_krb5.so.2 => /usr/lib/libgssapi\_krb5.so.2 (0xb7cf3000)  
libkrb5.so.3 => /usr/lib/libkrb5.so.3 (0xb7c6b000)  
libk5crypto.so.3 => /usr/lib/libk5crypto.so.3 (0xb7c46000)  
libcom\_err.so.2 => /lib/libcom\_err.so.2 (0xb7c43000)  
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7af8000)  
/lib/ld-linux.so.2 (0xb7f11000)  
libsepol.so.1 => /lib/libsepol.so.1 (0xb7ab7000)  
libkrb5support.so.0 => /usr/lib/libkrb5support.so.0 (0xb7aaf000)  
libkeyutils.so.1 => /lib/libkeyutils.so.1 (0xb7aad000)

# Repository

- I pacchetti possono essere scaricati e gestiti singolarmente
- Normalmente però si usano i repository (repo)
  - raccolte indicizzate di pacchetti
  - possono essere online o su filesystem locali
- I **package manager** leggono per ogni repo l'indice e i metadati dei pacchetti
  - conoscono quali versioni sono disponibili per ogni pacchetto
  - conoscono le dipendenze tra pacchetti (e quindi come risolverle)
- Collocazioni delle liste di repo ed esempi:

- `/etc/apt/sources.list` `/etc/apt/sources.list.d/*`  
`deb http://archive.ubuntu.com/ubuntu bionic-updates universe`
- `/etc/yum.conf` `/etc/yum.repos.d/*.repo`  
`[base]`  
`name=CentOS-$releasever - Base`  
`mirrorlist=http://mirrorlist.centos.org/?`  
`release=$releasever&arch=$basearch&repo=os&infra=$infra`  
`#baseurl=http://mirror.centos.org/centos/$releasever/os/$basearch/`  
`gpgcheck=1`  
`gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-7`



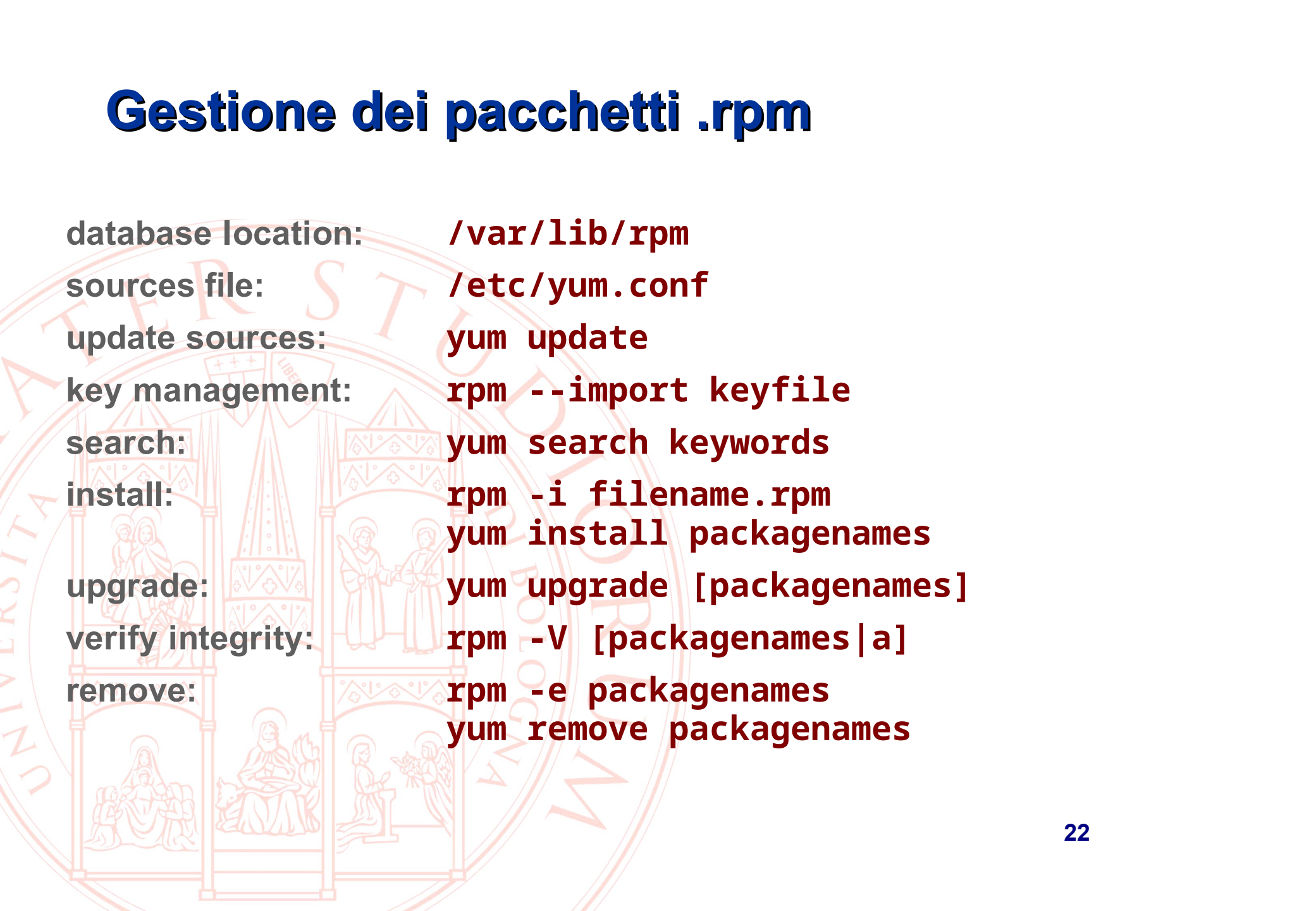
# Gestione dei pacchetti .deb



|                    |  |
|--------------------|--|
| database location: | <code>/var/lib/dpkg, /var/lib/apt</code>                                       |
| sources file:      | <code>/etc/apt/sources.list</code>   |
| update sources:    | <code>apt-get update</code>  |
| key management:    | <code>apt-key</code>   |
| search:            | <code>apt-cache search keywords</code>   |
| install:           | <code>dpkg -i filename.deb</code><br><code>apt-get install packagenames</code> |
| upgrade            | <code>apt-get upgrade [packagenames]</code>                                    |
| remove             | <code>dpkg -r packagename</code><br><code>apt-get remove packagenames</code>   |

(i suffissi **-get** e **-cache** possono essere omessi nelle distribuzioni più recenti, in cui il comando **apt** regge tutti i sotto-comandi come **search**, **update**, **install**, ...)

# Gestione dei pacchetti .rpm



|                    |   |
|--------------------|---|
| database location: | <code>/var/lib/rpm</code>   |
| sources file:      | <code>/etc/yum.conf</code>  |
| update sources:    | <code>yum update</code>   |
| key management:    | <code>rpm --import keyfile</code>   |
| search:            | <code>yum search keywords</code>  |
| install:           | <code>rpm -i filename.rpm</code><br><code>yum install packagenames</code> |
| upgrade:           | <code>yum upgrade [packagenames]</code>                                   |
| verify integrity:  | <code>rpm -V [packagenames a]</code>                                      |
| remove:            | <code>rpm -e packagenames</code><br><code>yum remove packagenames</code>  |

# Verifica dell'autenticità

- La firma dei pacchetti è gestita centralmente
- I *maintainer* di una distribuzione forniscono le chiavi di verifica nei media di installazione ufficiali o sui *repository* online
- I set di chiavi possono essere gestiti in modo standard con GnuPG  
es. `.deb`-based mettono gpg keyrings in `/etc/apt/trusted.gpg.d/`
- ... ma è più comune usare strumenti forniti dalla distribuzione

```
.deb apt-key {add file | list | del keyid | adv --recv-key keyid | ... }
```

```
.rpm rpm {--import | -e | -q[ai] | ...}
```

rpm tratta le chiavi come se fossero pacchetti  
→ si possono usare gli stessi  
comandi per interrogarli, eliminarli, ecc.

# Lavorare coi repository

- Un'esigenza molto comune è quella di installare software ben supportato ma non incluso per qualsiasi motivo nei canali ufficiali della distribuzione
- Si aggiunge semplicemente il repository all'elenco

- Apt (deb):

- `/etc/apt/sources.list.d/virtualbox.list :`

- `deb http://download.virtualbox.org/virtualbox/debian xenial contrib`

- Yum (rpm):

- `/etc/yum.repos.d/epel.repo :`

- `[epel]`

- `name=Epel Linux -`

- `baseurl=http://mirror.example.com/repo/epel5_x86_64`

- `enabled=1`

- `gpgcheck=0`



# Gestire la provenienza dei pacchetti

- Si può generare confusione se un pacchetto con lo stesso nome è presente in versioni diverse in repository differenti
  - I package manager, di default, scelgono sempre la versione più avanzata
  - supponiamo di aver aggiunto un repo semisconosciuto per installare un'applicazione innocua
  - se a tale repo viene aggiunto un pacchetto "core" dichiarato più recente della versione ufficiale → **software injection!**
- In alcuni casi anche aggiornamenti nello stesso repo sono indesiderabili
  - situazioni legacy
- La situazione va controllata e gestita
  - Controllo della provenienza di un pacchetto
    - Yum: **repoquery -i [package name]**
    - Apt: **apt-cache showpkg [package name]**
  - Elenco dei pacchetti provenienti da un repo
    - Yum: **yum list installed | grep [repo name]**
    - Apt: vari comandi per estrarre manualmente info dai file della cache

# Limitare le modifiche automatiche

- Per evitare a priori problemi in sistemi con dipendenze complesse (ad esempio mix di pacchetti installati manualmente e via package manager)

- Version locking/pinning

- Apt

- editare `/etc/apt/preferences.d/*`

- <https://wiki.debian.org/AptPreferences>

- Yum

- `yum install yum-plugin-versionlock`

- poi

- `yum versionlock [package name]`

- o editare a mano

- `/etc/yum/pluginconf.d/versionlock.list`

# Build your own repo (rpm)

- I package manager sono molto utili per "tenere in ordine"
- È sconsigliabile mischiare installazioni manuali con pacchetti
- Non è difficile pacchettizzare le proprie applicazioni!
- Nel mondo RPM
  - si configura un ambiente di build per un utente (non root!)
  - si preparano i sorgenti e tutti i file che devono essere inclusi in un pacchetto
  - si effettua il build del pacchetto
  - lo si carica su di un server web
  - si generano gli indici che rendono riconoscibile il sito come repository

# Build your own repo (rpm)

- Per il build, si configura un ambiente per un utente (non root!)

- file `~/.rpmmacros`

```
%packager      Marco Prandini <marco.prandini@unibo.it>
%vendor        DISI
%_topdir        /home/prandini/rpmbuild
%_signature     gpg
%_gpg_name      Marco Prandini (Unibo) <marco.prandini@unibo.it>
```

- devono essere presenti alcune cartelle sotto `_topdir`

|               |  |
|---------------|--|
| <b>SPECS</b>  | contiene i file di specifica dei pacchetti |
| <b>SOURCE</b> | contiene i sorgenti da compilare/includere |
| <b>BUILD</b>  | contenitore per il pacchetto "aperto"      |
| <b>SRPMS</b>  | destinazione dei pacchetti sorgente        |
| <b>RPMS</b>   | destinazione dei pacchetti binari          |



# Build your own repo (rpm)

- Un file di specifiche contiene tra le altre cose
  - elenco dei sorgenti **Sources** o **Sources0**, **Sources1**...
    - devono essere in SOURCES, in formato .tar.gz, e contenere una directory di primo livello con lo stesso nome del file
  - **Requires**
    - indica le dipendenze esplicite, rpm fa da solo l'elenco delle dipendenze implicite (esamina i binari ed include tutte le librerie necessarie a lanciarli)
  - **BuildRequires**
    - indica i pacchetti che servono per fare il build del pacchetto
  - **BuildRoot** è dove il build viene eseguito (possibile fonte di rischio visto che è parametrizzato)
- Si usa **rpmbuild -ba --rmsource --sign test.spec**
  - genera **RPMS/architettura/test-1.0-1.noarch.rpm** e **SRPMS/...**
  - rimuove i sorgenti originali
  - per ripristinare i sorgenti basta reinstallarli con **rpm -Uh SRPMS/test...**
- Si copia il file rpm sul server web
  - ad esempio in **/var/www/repos/myrepo/RPMS**
  - il repo sarà visibile dai client con **baseurl=http://server/myrepo** e vi si lancia
    - **createrepo /var/www/repos/myrepo/RPMS**

# deb e rpm

## ■ Link per deb

<http://www.debian.org/doc/manuals/debian-reference/ch02.en.html>

[http://guide.debianizzati.org/index.php/Introduzione\\_all'\\_Apt\\_System](http://guide.debianizzati.org/index.php/Introduzione_all'_Apt_System)

## ■ Link per rpm

<http://yum.baseurl.org/wiki/YumCommands>

<http://yum.baseurl.org/wiki/RpmCommands>

# Problematiche di aggiornamento

- Quando si aggiorna un pacchetto software già in uso sul sistema, si deve tener conto di potenziali problemi derivanti da:
  - **prerequisiti**
    - pacchetti che devono esistere perchè il candidato funzioni bene
    - come nel caso dell'installazione
    - potrebbe non essere facile aggiornare i pacchetti-prerequisiti senza causare problemi ad altri software che li utilizzano
  - **configurazione**
    - eventuali modifiche incompatibili apportate al formato delle direttive di configurazione già messe a punto per la versione funzionante

# Problematiche di aggiornamento

## ■ (continua)

- dipendenze di altri software e test di non regressione
  - modifiche apportate alle interfacce o alle funzionalità del software potrebbero influire sul funzionamento di altri software
  - *configurazione del PATH* per impostare l'ordine di ricerca degli eseguibili nelle directory
  - predisposizione di configurazioni di test per far coesistere le due versioni durante le fasi di verifica
    - es. binding a socket, porte, IP diversi --> problemi di trasparenza per l'utente, licenze, configurazione delle controparti se il software da testare interagisce attraverso interfacce standard



# Problematiche di aggiornamento

- (continua dipendenze e test)
  - *configurazione del loader* per far convivere differenti versioni di librerie dinamiche, si veda la man page `ld(1)`, specialmente le sezioni sui parametri `-rpath` e `-rpath-link`
    - modifica dei settaggi di default in **`/etc/ld.so.conf`** (da applicare con **`ldconfig`**)
    - uso delle variabili `LD_LIBRARY_PATH` in fase di loading e `LD_RUN_PATH` in fase di linking
    - Es:

```
# ldd /usr/sbin/sshd
...
libz.so.1 => /usr/lib/libz.so.1 (0xb7e0c000)
...
# export LD_LIBRARY_PATH=/usr/local/lib
# ldd /usr/sbin/sshd
...
libz.so.1 => /usr/local/lib/libz.so.1 (0xb7dab000)
```

# Disinstallazione

- Presenta gli stessi problemi dell'aggiornamento in termini di eventuale dipendenza di altri software da quello che si sta per rimuovere
  - in entrambi i casi può essere molto difficile prevedere gli effetti sul sistema se la gestione è manuale
  - il *grafo delle dipendenze* è quindi il valore aggiunto più significativo dei sistemi a pacchetti
  - può essere molto utile sfruttare la possibilità offerta dai package manager di creare i propri pacchetti, per gestire il software installato manualmente tramite il sistema automatico di verifica delle dipendenze (ma ciò significa censirle con precisione all'installazione)

# Snap packages

- Pacchetti software che devono essere
  - particolarmente longevi
  - basati su funzioni non standard di una distribuzione
  - distribuiti in modo indipendente dai canali della distribuzionepossono essere composti come *snap*

<https://tutorials.ubuntu.com/tutorial/create-your-first-snap#0>

<https://tutorials.ubuntu.com/tutorial/basic-snap-usage#0>

# Virtual environments (VE)

- Le macchine virtuali rispondono a necessità comuni
  - preconfigurare sistemi
  - isolare diversi ambienti di esecuzione
  - definire limiti di uso delle risorse
- Spesso sono eccessivamente pesanti
  - impongono l'installazione di un intero SO
    - inutile replicazione di librerie e utilità
    - overhead di memoria in esecuzione
  - limitano l'accesso *bare metal* alle risorse
- Soluzione: isolare l'ambiente di esecuzione di (gruppi di) processi condividendo il sistema operativo
  - l'isolamento non è totale
  - non si possono eseguire applicazioni incompatibili col sistema
  - l'accesso all'hardware è diretto
  - le risorse condivise non sono replicate
- OpenBSD Jails, Solaris Zones, Linux namespaces



# Cgroups, namespaces, unionfs

- Il kernel fornisce tre funzionalità a supporto dei VE
  - **control groups (cgroups):**
    - misurano e limitano la quantità di risorse che un processo può consumare
      - memoria
      - CPU
      - I/O
      - rete
    - controllano l'accesso ai device (via /dev)
  - **namespaces:**
    - mostrano a un processo istanze di risorse fisiche condivise
    - un processo in un namespace non vede la risorsa fisica reale
    - l'istanza appare al processo come a suo uso esclusivo
    - ogni processo vive in un namespace di ogni tipo tra
      - mnt (mount points, filesystems)
      - pid (processes)
      - net (network stack)
      - ipc (System V IPC)
      - uts (hostname)
      - user (UIDs)
  - **union-capable filesystems:**
    - combinano due supporti per mostrare un FS unico (es. snapshot)

# Containers

- Usando il partizionamento di risorse, si possono definire i *containers*
  - definiscono in modo coerente cgroups e namespaces
  - specificano come esporre le risorse viste dal processo interno al sistema
  - includono il software necessario all'esecuzione del processo
- La gestione dei container è semplificata da strumenti come
  - LXC (Linux Containers)
  - Docker
  - CoreOS rkt
  - Apache Mesos